14

Some Observations on Conceptual Programming

Erik Sandewall

Informatics Laboratory
Linkoeping University, Sweden

One fairly large LISP program is analyzed carefully with respect to data structures, program structure, and implications on programming methodology. The program does a fairly conventional data-base management job, and was originally written for its practical purpose, rather than as a methodology experiment. The analysis is performed in the framework of conceptual programming, which says, e.g., that information about program structure and about data structures (for example the information commonly found in declarations) should be stored in the user-accessible data base of the programming system, and be usable in multiple ways. The long-range goal is to enable the programming system to "understand" the program, and to communicate with the programmer on his natural conceptual level. The analysis of the sample program results in suggestions for a number of programming methods, for example to use a second-order declaration structure as the top-level structure of the program, and then associate pieces of code with entities in the declarations, or structures formed from them. Another suggestion is to distinguish between an "execution" model and an "initialization" model of the system. The former is basically a combined data-flow and procedure-call structure; the latter is an idealized "program" for the conduct of an interactive session, plus a collection of inserts into or updates of that program.

INTRODUCTORY REMARKS

The work reported here consisted in a detailed analysis of an existing program of non-trivial size, resulting in a number of suggestions for program structure and program development methods. Since they are only based on observations in one program, the suggestions have the status of hypotheses rather than proven facts, and additional verification or falsification is desirable (and intended).

A study of this kind could probably not have been performed without a certain "ideological" background. One cannot observe merely by looking; one needs a conceptual framework in order to select what to look for, and to interpret what one sees. In the present study, my framework was the long-range ideal of computer-supported program development. I envisage a situation, at least in the limit, where the primary representation of the computer program is as a structure in the computer. The user would only in exceptional cases see the whole program, i.e., a dump of the whole structure. Usually he would build it up piecewise, and inspect it piecewise by requesting that specific sections or projections of the structure be printed out. The programming system should "understand" the program structure well enough that it can communicate with the user about the program in a mature way, using the same conceptual level (although not necessarily the language) that two programmers would use between themselves.

This ideal is sufficiently vague and remote that it should be considered as a direction of movement, rather than a specific goal that one will reach and prove that he has reached. I adopted it as an ideal for two reasons:

- It is the only possible way to make the computer support the programming activity. There is a clear need for very-high-level programming systems, which help the programmer (or a team of programmers) keep track of large and complex programs. Winograd (Winograd, 1974) has successfully argued this point. At the same time, since a program is a richly connected structure of goals, decisions, conventions, solutions to sub-problems, and compromises, one cannot expect a programming system to understand one part of that structure if it does not have access to the other part. The system must have a full understanding of the program, or it will not have any useful understanding of it at all. Also, it seems to me that the only possible way of entering that structure into the computer is either to let the computer generate it itself (i.e., fully autonomous automatic programming) or to let the programming system participate in the program development under the guidance of the programmer. The first alternative is very remote, and the latter alternative is somewhat more realistic. The second alternative might also be a good subgoal for the first one, although that is not part of my motivation.
- The second reason is that, by developing an appropriate structure for the representation of a program in the above mentioned type of programming system, one is forced to make a precise analysis of structures which would otherwise just be intuitively understood. In designing such a "conceptual program structure," one is encouraged to ignore all the many trivial aspects of textual representations and syntactic sugaring, and focus on what are the essential structures in programs.

The term *conceptual programming* is chosen for the proposed programming style, where the program is represented as a "conceptual structure" in the programming system with which the programmer interacts for program development. This "ideal" programming style is not taken entirely out of the blue. First, the LISP programming language (and its cousin, the Vienna definition language) may be viewed as first steps towards that ideal. More important, several widespread although previously undocumented practices of LISP programming take additional steps in the same direction. Similar development can and does take

place in the framework of other languages, although more slowly and with much greater difficulty.

The conceptual programming ideal has been used as the framework for the program analysis that is reported here. Conversely, the present case analysis resulted in concrete suggestions for program organization and programming methodology, which make the conceptual programming idea(l) more specific in at least some ways.The sample program, written by Dave McDonald at MIT, uses those current programming practices which I view as steps in the direction of conceptual programming. The study resulted in some results regarding the extent and limitations of present practices, but more important, it generated a number of concrete suggestions about how these methods could be extended and improved.

This work was done in the context of the programming language LISP, a language that is used intensively among some groups of researchers, but rejected or shrugged off by many others. I believe that this is largely because all available textbooks on LISP are bad, and describe the language from an uninteresting and irrelevant point of view. Part I tries to make up for that by presenting what in my view are the significant properties of modern LISP system, for the benefit of readers who are not immersed in the LISP culture. Part II describes and discusses the inspiration of this work, that is it describes the current programming practices, and extrapolates to the long-range ideal. Part III reports on the detailed study of McDonald's program, and is the main section of the paper. Part IV is short, and attempts to summarize the results as a list of specific findings.

PART I

LISP AS A BASIS FOR CONCEPTUAL PROGRAMMING

LISP systems have certain properties which make them suitable as an environment for conceptual programming, which explains why the conceptual-programming trend has developed in the LISP-using community. Other languages such as SNOBOL or APL have the same properties and could presumably also be used for conceptual programming. Languages such as PL/I, Pascal, or Simula 67 lack the combination of those properties, and could not be used except after non-trivial modification. The purpose of the present section is to describe those properties which are essential for conceptual programming.

I think of LISP as determined by three basic design decisions. The first two are:

(a) for ease of debugging, the system shall be *incremental*, meaning that the programming system performs a read-evaluate-print loop, where in each cycle the user enters an expression, has it evaluated, and sees the result. The expressions may serve to define a procedure, store something in the data base, evaluate an expression in order to test a procedure, or edit a procedure or the data base. The user communicates all the time with one single programming system, and does not have to switch between "edit", "compile", and "execute" modes.

(b) because of the intended applications, the language shall contain facilities for handling *data structures* and maintaining a *data base*.

Neither of these criteria is unique: APL and many implementations of BASIC satisfy the first criterium, and PL/1, Algol 68, Pascal, etc. satisfy the second criterium. However, *the combination of these two purposes is not trivial* to achieve. The reason is that the read-evaluate-print loop assumes that one can type in arguments of procedures to the programming system, and obtain their values typed out on the console. In order to also satisfy the second requirement input and output of data structures must then be defined—which it is not, in conventional programming languages.

In order to account for input and output, LISP encourages a different method of data structuring than record-oriented languages. Consider the traditional example of family relationships: suppose one wants to design a data base that maintains information about persons, and in particular, information about parent-child and brother-sister relationships. In record-oriented languages, the following structure is natural: each person is represented as a record, with pointers to other records, for example a "father" pointer and a "mother" pointer. Also, the circumstance that one person may have several children is handled by letting each person point to its "oldest child", and also to let each person (namely each child) point to its "next younger sibling". This is illustrated in Figure 1.

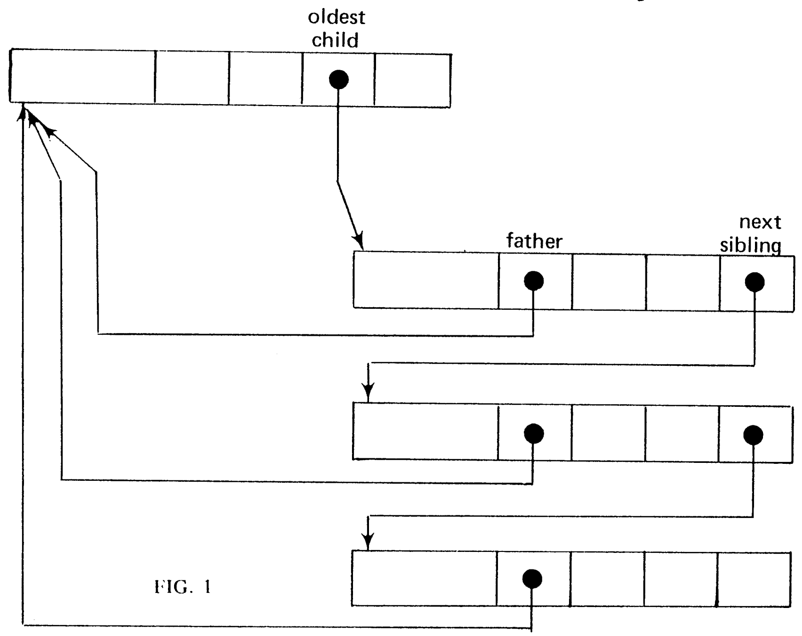

FIG. 1

In LISP, one is encouraged to use a built-in data type called an *atom*, i.e., a special kind of record which stands in a one-to-one relation with a character string. Thus each time the standard *read* routine encounters that character string, the same record is retrieved. In the present example, if JOHN's children are called BOB, DICK and MARY, and ignoring the problem that several persons may have the same name, one would have one atom for each of these names. Furthermore, the system contains two primitive operations, which we shall here call *get* and *put*, and which serve the following purposes: *put* is called with three arguments, and stores a property assignment in the data base, for example

put(BOB,FATHER,JOHN)

with the obvious intended meaning (John is Bob's father). Similarly, the function *get* retrieves a property from the data base, for example

get(BOB,FATHER)

which should return the value JOHN (i.e., the atom = the record which stands in a one-to-one relationship with the character string "JOHN").

So far there is no conceptual difference from the structure in Figure 1. The difference comes when handling the set of children of a person, where the LISP-oriented structure is to form the sequence of the children,

<BOB, DICK, MARY>

and to assign that sequence as a property:

put(JOHN,CHILDREN,<BOB,DICK,MARY>)

This sequence has traditionally been implemented using binary pairs of pointers, as shown in Figure 2. (More storage-efficient representations are presently being developed.) Thus the conventional record structure encourages one to represent sets and sequences of objects by *threading through* them, giving the structure of shark's teeth on a necklace, whereas the LISP-oriented structure looks like a comb with pointers *down* to the objects involved.

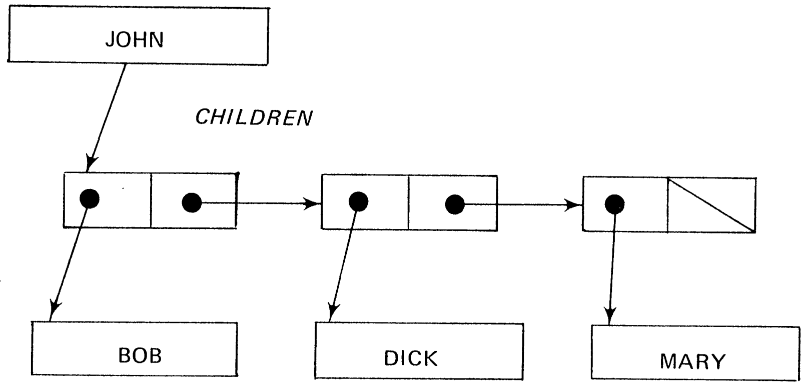

FIG. 2

It is not my purpose here to argue for one or the other of these representation models; they both have their merits and demerits. Let me point out, however, that it is not trivial to use the LISP data structure in conventional record-oriented languages. The structure of pairs of pointers can of course easily be implemented, and one can also write a program which converts a character-string to a corresponding atom, or uniquized record, by going through a symbol-table. However, that only accounts for atoms that appear in input data to the system, but not for atoms that appear as constants in the program itself. In many cases it is important to be able to use the same atom in the program as a constant, and in input data.†

The third design consideration for LISP is that there should be a standard convention for *representing programs as data structures in the language*, in order to facilitate generation and manipulation of programs.

The appropriate representation for a program is of course that of a tree. Thus the expression a + b x c + d is viewed as the tree in Figure 3:

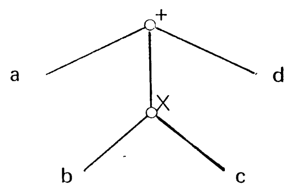

FIG. 3

which is encoded as the following data structure in Figure 4.

The boxes in Figure 4 that contain character strings are realized as atoms in the executing programming system, and in general, all atomic components of a program (procedure names, entities in declarations, variables, etc.) are represented as atoms. The atoms PLUS and TIMES stand of course for + and x.

For purposes of program analysis and generation, the data structure representation of the program is more convenient than the conventional representation as a character string, since the components of the programs (identifiers, and sub-expressions on different levels) have already been extracted, and are available as entities. The internal data-structure representation of the program is similar to an intermediate representation in a conventional compiler.

This program representation goes well with the first two design decisions. Since procedures are defined and edited by evaluating expressions in the language, it follows that these operations do not *have to* be performed on the top level of interaction with the system; they may also be performed during the execution of a program. Experienced LISP programmers very often use that possibility for low-key program generation. Also, since all entities in the program are atoms, it becomes possible to associate information with program entities, for example to store descriptions of the data structure (the declarations in the program) in the data base. It becomes possible to write programs that inspect their own declarations, and each user can store arbitarary information (for example, for documentation purposes) with the declarative information.

†The situation is analogous to the handling of numbers in algebraic programming languages. A Fortran system allows input data to contain characters strings such as "4.65", and converts such a string to the internal representation in the computer, usually as the floating-point contents of a cell. It also allows such expressions to appear in the program, and if so, internizes them at compile time and saves them in such a fashion that they are accessible when the program is executed. An analogous mechanism for atoms is necessary in order to incorporate them into a compilation-oriented programming system.

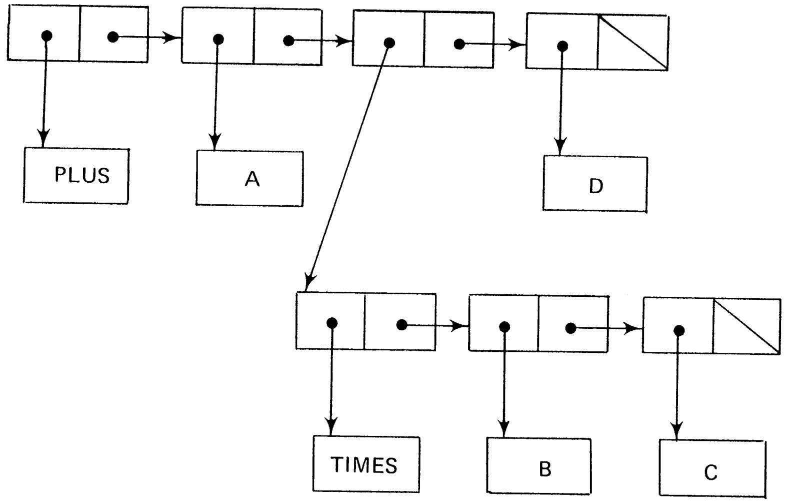

FIG. 4

The question of input of programs in LISP is often misunderstood, and shall therefore be discussed here. The data-structure representation of the program is the preferred internal representation, meaning that the interpreter and the compiler are defined to operate on it. There is, however, no commitment as to how the structure is to be entered. One possibility is to have an Algol-like language which is translated to that structure, and which may then have declarations, infix operators, **for** statements, and so forth. Translators for several such input languages have been developed. However, since input and output are defined for arbitrary data structures, one may also use that facility for entering and printing out programs. Using standard data-structure I/O, the above expression would be represented on paper as

<PLUS A <TIMES B C> D>

Many users prefer that notation, since it provides a more direct contact with the internal representation of the program, but the question of external program notation is a matter of personal taste, not basic system design.

The two data types of atoms and binary pairs of pointers, are basic for the language, but they are not the only allowed types. The two basic types have a special status since the internal representation of programs uses them, and since input/output for them is predefined, but modern LISP systems such as INTERLISP (Teitelman, 1974) and LISP derivatives such as EPL (Wegbreit, 1972) contain facilities which enable the programmer to use other data types as well, and also to define rules for input, output, and evaluation of his own data types.

The basic operations *get* and *put* that were described above can easily be generalized. In essence, *put* associates an arbitrary expression with a pair of atoms, for example associates JOHN with the bituple <BOB, FATHER>. It is trivial to write a more general function *put** which associates expressions with arbitrary nested tuples, and not just with pairs of atoms. Modern LISP systems contain a number of such functions, implemented using either a tree or a hashing technique (or a combination of those) for storage of the associations.

The *put* operation first appears to be analogous to the assignment of terms in a record, when a conventional record structure is used. There are however several important differences. First, both the first and the second argument of *put* are arbitrary atoms, and the program may variablize with respect to either of them, or both. In a conventional language, if one has an assignment statement of the form

```
x.father := readstring();
```

then *x* is a variable (and may be a whole expression), but *father* is a constant. In the corresponding situation using the *put* function, one may have a variable that happens to be bound to *father*, or an expression that evaluates to it, or a loop of the type

```
for p in list-of-property-names do
    begin
    print(p);
    x.p := readvalue()
    end
```

Another difference is that the *put* operation is dynamic, i.e., the basic programming system does not maintain declarations that specify which properties are allowed for which objects. This is significant in an incremental environment. Suppose one is interacting with the programming system, and he has loaded a large program, and a data base which is being used when testing the program. He now decides to add one more property to a type of objects. With a LISP-style *put* function, he just goes ahead and stores those properties. If instead the properties had been frozen by declarations, he would have to:

- dump the data base (since the data base has probably been input interactively and incrementally during the testing of the program);

- edit the program;
- recompile the program;
- load the data base.

The same advantage is evident if another program performs the change of structure.

Thus the basic programming system does not maintain or use declarations, but at the same time, the properties of the system *encourage* the development of higher-level systems that use declarations. In the above example with family relationships, one might store in the data base

```
put(JOHN,TYPE,PERSON)
put(PERSON,PROPERTIES, <FATHER,MOTHER,CHILDREN>)
```

indicating that JOHN has type PERSON, and which properties are expected for that type. One may also store

```
put(FATHER,STRUCTURE,PERSON)
put(CHILDREN,STRUCTURE, <SEQUENCE PERSON>)
```

to specify the desired structures of such properties. Even in these very simple examples, one makes use of the fact that program entitites, such as type names and property names, can also be used as data. The fact that programs are represented as data structures facilitate the task of writing programs that check a program's consistency with its declarations. (Nordström, forthcoming, is doing that with Simula 67 as the input language).

More fancy declaration systems are possible and worthwhile, such as declarations which specify how other declarations are stored. A major theme of the present paper is how such declarative structures should be developed.

PART II

CONDENSED IDEAS ABOUT CONCEPTUAL PROGRAMMING

The ideas of coneptual programming are very much "in the air" in the artificial intelligence community. This section represents an attempt to "condense" some of those ideas, in order to set the stage for the report on actual work in the next section. The methods and ideas that are described here largely represent my own experience from working with programs such as PCDB (Sandewall, 1971,1973) and REDFUN (Beckman, *et al.,* 1975), but that experience closely parallels the ideas and experience of many others. It is hoped that many readers will experience a sense of *déjà vue* when reading this section.

Data-driven programs

A classical model of a program is that it is a collection of procedures which call each other. Each procedure has a name, and another procedure can call it by explicitly mentioning its name. The calling structure is statically available, so for example it is possible to write programs which take a set of procedures and produce a graphical representation of the calling structure.

In data-driven programs, on the other hand, the procedure calls are indirect via the data base. One ("executive") procedure or program accepts input data, either from user input or as arguments, retrieves procedures which have been associated in the data base with data items that appeared in input, and executes those procedures. This indirect calling structure is illustrated in Figure 5.

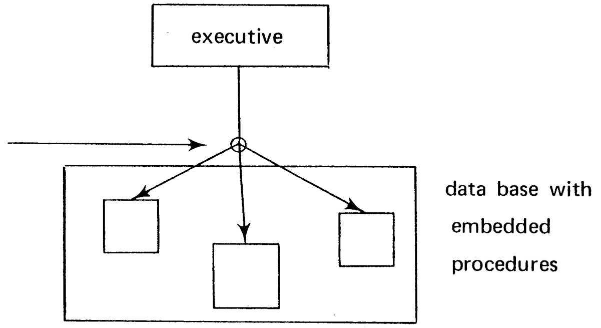

FIG. 5

There is an abundance of examples of this technique. The PCDB system (Sandewall, 1971,1973), maintains a data base of assertions in predicate calculus, and assumes that each relation symbol is associated with a storage procedure (for storing the relationship in the data base), a retrieval procedure (for looking it up), a search procedure (which uses deduction to look it up), procedures for answering open questions, etc. In order to assert a relationship such as

COMPONENTS(finger,hand,5)

to the system, one calls a general-purpose procedure *store* with the arguments

store(COMPONENTS, FINGER, HAND, 5)

where store(r,x,y . . .) is defined to look up get(r,STOREFN) and execute it with (x,y . . .) as its arument list. Thus *store* makes an indirect call, or *dispatches* to procedures associated with relation names.

Programs that operate on LISP programs provide several examples of data-driven-ness. The REDFUN program (Beckman, *et al.,* 1975) performs partial evaluation and other simplification of LISP procedure definitions. It allows that procedure names in the program that is to be simplified, may be associated with specialized procedures which know how to simplify expressions with that procedure as its leading operand. Thus for a trivial example, the simplification procedure for PLUS would embed the knowledge that a + 0 = a. Prettyprinting programs (i.e., programs which produce nicely indented presentations of programs)

such as the one in MACLISP (Moon, 1974) are data-driven with respect to procedure names in a similar fashion. Risch (Risch, 1975) discusses a number of program-manipulating programs that dispatch on procedure names, and proposes a way of systematizing their conventions so that they can dispatch to the same set of procedures.

In the continued discussion we shall repeatedly use the same example, namely a program system written by Dave McDonald at the M.I.T. Artificial Intelligence Laboratory. The program maintains a data base of document descriptions, such as author, title, year of publication, and so on for several types of documents (books, articles in journals, internal research memoranda, etc.). The major purpose of the system is to take a list of document identifiers, and to print out the list of the author, title, etc. of those documents, in a format which is suitable for the list of references at the end of an intended new paper. In particular, the printout program can be instructed to conform with any of the idiosyncratic sets of rules that different journals impose on authors (first name of author before or after last name; names of journals must or must not be abbreviated; and so on). The system is called the bibliography system.

The system maintains an active data base of document descriptions, as a data structure in the LISP system. The *printout program* draws on that data base in preparing the printout. The system also includes a number of other programs: a *data entry program* that prompts the user for contributions to the data base, a *saving program* that transfers the active data base to a "passive" representation as a text file, and a *re-creation program* that reloads the active data base from one or more such text files. Finally, the data entry program continuously adds the user's input to another text file as a safeguard against the eventuality of system breakdown, and a *recovery* program reloads the text file in that event.

Thus the topmost structure of the system is that there are a number of "data pools", and a number of programs which transfer data between these data pools. (That is of course a third way of program-to-program communication, besides direct calls and data-driven calls). The data flow structure is illustrated in Figure 6.

In the active data base, each document is represented as an atom, with associated properties for author, title, etc. The atom which serves as document i.d. actually looks like B135, i.e., it is essentially a number. Also, each document id has a property which specifies its exact type, which can be either of (presently) six atoms such as BOOK, JOURNAL-ARTICLE, REPORT, etc. Finally, the type name is associated with information as to what properties objects of that type can have: all documents have an AUTHOR and a TITLE property, but only articles in journals and collection volumes have an associated page number.

The major programs in the bibliography system are organized approximately as follows: on the top level there is a loop over a number of document descriptions. In each cycle of the loop, the program determines the type of the document, by prompting the user (in the case of the data entry program), by looking up its TYPE property, or by having it available in the computational context. It then makes a loop over the names of properties that objects of that type can

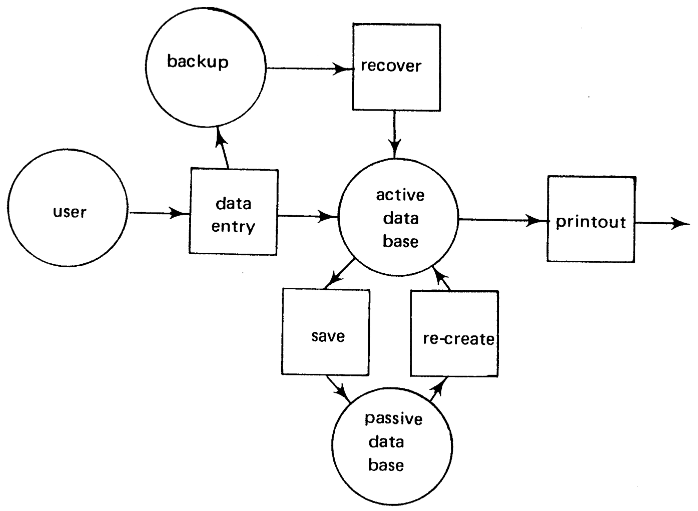

FIG. 6

have, and for each property-name, calls a procedure which is associated with it. Thus a property-name such as AUTHOR is associated with one READFN for prompting the user about that property, one PRINTFN for printing it out, and so on.

I have here idealized the structure for the purpose of simplicity. In actual fact, the "procedure" is often a structure which contains both a procedure name and a number of flags or parameters which have a special significance in the performance of the task. Also, not all programs proceed through the type name to get to the property names: it is sometimes possible to go directly from the document name to the set of properties that it has, make a loop over the existing properties, and call the appropriate procedure for each of them. Such variations to the theme will be further discussed in Part III of this paper, but they indicate that there is a lot of freedom in how data-drivenness is implemented. Data-driven-ness is a style of programming, not one primitive operation (although certain primitive facilities are necessary for doing it). The data-driving or dispatching mechanism is very similar to indirect addressing in machine language. The **case** construction in higher-level languages serves some of the same purpose, but the dispatching mechanism has the advantage that additional cases can be added dynamically during an interactive session with the programming system, or by another program.

Just as the LISP function *get* can be generalized from the case where a pair of atoms is associated with an expression, to the case where the argument is an arbitrary atom or nested tuple, so also the data-driven procedures can be associated with arbitrary expressions, and not just pairs such as <AUTHOR, READFN>.

The method of data-driven programs is a sound programming practice for a number of reasons. Most of them have been discussed in a previous paper (Sandewall, 1975), but I shall shortly reiterate them here.

Sound naming. Conventional procedures are given names which are "mnemonic" in the sense that when one sees the name, he may get some feeling for what the procedure does. The reverse is however usually not true: if one looks for a procedure that does a certain thing, he is probably not able to guess its exact name. Data-driven procedures are characterized by expressions which are combinations of two or more "names", for example the just discussed <AUTHOR,READFN>. Given some conventions which hold throughout the user's program, such a combination can uniquely specify the purpose of the procedure.

The following example illustrates the point. A few years ago, I wrote a semantic-data-base program, which performs deductive storage and retrieval of typical natural-language expressions (kernel sentences, property assignments, time and space information, etc.) in a data base. The program was data-driven and organized around a number of predicate-calculus relations and functions, about 30 altogether. It used the PCDB structure described above, so for each relation and function, there were a number of procedures for storage of the relationship in the data base, retrieval if the relationship is explicitly stored, retrieval by deduction for open and closed questions, etc. There were altogether about 150 procedures with an almost arbitrary calling structure–one procedure called several others, and there was no visible clustering in the calling structure.

Normally a collection of 150 arbitrary procedures would be fairly difficult to keep track of and update. In this case however, each procedure was characterized by its relation name, its purpose (storage, retrieval, etc.), and in the case of open questions, which argument position(s) were being asked for. As a result it is trivial to find the procedure which performs a given task, and when the program is to be modified there is rarely any question as to where the change is to go, and whether the change may obstruct other parts of the program. And most important, this is achieved without separate documentation of all those procedures–one concise description of the naming conventions is sufficient. Another reason for the clarity of the organization in that program, is that the underlying predicate-calculus representation provided a structure around which the program could be built. One may view it as a relatively problem-oriented description of the task (more problem-oriented than the actual program, that is), and pieces of program were then associated with items in the task description, namely relation and function names.

If the only purpose is to assign structured procedure names, then that may of course also be achieved by "hyphenated" names. Thus the procedure which for a given x determines the y for which SUBPART(x,y) holds, could be called SUBPART_SEARCH_2(x). But it is preferable to let the components of the name be separate entities, so that one can associate information with each constituent (for example with relation names), and also so that driving procedures can keep one part constant and let the other be variable. This is used in the aforementioned definition of *store* as

store(r,x,y . . .) = get(r,STOREFN)(x,y, . . .)

and of course in many cases in the bibliography system.

Provides an extensible input language. When a program is data-driven using input from the user, the name of each data-driven procedure uses terms which appear in the input language. Thus the agreement between the terminology of the application and the naming in the program, is maximized. This is a desirable practice just for the purpose of manual program maintenance, but it offers the additional possibility of associating both a description of the task environment, and the program for performing a part of that task, with the same atom or "node" in the data base, which makes it easier to check them against each other, or to generate the program from the description. It also makes it maximally easy to extend the input language: new terms are added by adding to the data base a procedure which accounts for that term (of course within the limits and the framework of the executive level of the data-driven program).

Embedded, specialized programming languages. Interpreters, which inspect a program and execute procedures associated with operators in the language, are a special case of dispatching programs. It is common practice in LISP to set up specialized "languages" for specific purposes. For example, the INTERLISP system (Teitelman, 1974) enables the user to choose names for groups of procedures, global variables, and other global data that constitute a module, and which are to be printed out together as a text file. Each "file name" (module name) is then associated with a data structure which specifies how to print the file, for example which procedures shall go on it. In looking at such examples, one finds a spectrum from mere sets of parameters, to expressions in full-fledged programming languages, and the distinction is not particularly interesting. The point is that in order to keep such specialized programming languages small and simple, one wants to be able to call back to the host programming system from them, i.e., to reference procedures or code in the host language from the specialized language. Thus the specialized language can rely on the host system for the assortment of facilities that are always needed, such as file handling and interaction. Embedded languages in this sense seem as the best way to achieve the goal of "extensible programming languages", i.e., to enable each user to tailor the system to fit his needs.

The interpreters that implement such languages must make data-driven procedure calls. A number of programming-language mechanisms which have been proposed in recent years, such as pattern-directed invocation and demons, also rely for their implementation on the method of data-driven programs. Maybe the relative success of such systems is because they made the benefits of data-driven programming available to people who would otherwise not have used it. But it must be better to encourage the user to use the general method, than just provide him with specialized packages for a few operations.

Advising

The following practice is often used by LISP programmers, but may be less obvious to the user of another language. Suppose one wants to organize his program as a set of rules, each of which contains at least a criterium for when it is to be applied, and what one is to do then. Suppose in particular that the criterium for application can be characterized as a piece of data, for example a pattern. One wants to be able to associate several procedures with the same triggering datum, or invocation condition. Therefore, one lets the system create a program skeleton which is the default assignment to each triggering datum, and which may be for example an empty **begin - end** block (or in LISP terms, *progn* form). When the user enters a rule, usually during his interaction with the system, he uses a procedure which inserts the body of the rule in the appropriate place in that structure. In general, the skeleton provides the "glue" or the control structure which keeps the rules together. Thus several rules that trigger from the same datum can be entered at different times, and will be gradually assembled into a procedure.

Another advantage (besides the incrementality) is that the structure of the system as viewed by the user, may be different from the structure as viewed by the executing programming system. Several rules that trigger off the same datum must be kept together during execution, for obvious reasons, but when the user works with his program = set of rules, he may want to group them differently. Advising enables him to do exactly that.

Insertive programming

Both data-driven programming and advising serve the fundamental purpose of modularity: the program is split up into modules which are well named, which can be located in logically appropriate places, and whose interrelationships with their execution environment are well defined and understood. If the modules satisfy these requirements and if they are sufficiently small, such as one or few pages, then one does not have to be much concerned about their insides. Any competent programmer can go into such a small independent program, understand it, and modify it to his needs.

But it may not always be possible to reduce the problem to such small modules while retaining control of the relationships between modules, which raises the problem of program structure within a module, in order to allow larger modules. An obvious candidate is then the *hierarchical* program structure, as argued for example by (Dahl, 1972, p. 176 ff). I have some reservations about organizing an entire program system in a *uniformly* hierarchical fashion (for example, when every level is a sequence of "steps" which are decomposed as the next lower level), but it is clear that a uniform hierarchy in that sense is sometimes a powerful way of organizing a program.

The purpose of this section is to describe an extended program model which I shall call *insertive* programs (as compared to hierarchical programs), and which may be viewed as the generalization of advising to operate in the context of a hierarchical program. Insertive programs can best be introduced by means of the programming method that goes with them. This is natural since the hierarchical program structure is also associated with programming methods; they can be composed using a top-down method (successive decomposition) or a bottom-up method (successive agglomeration). Wirth (Wirth, 1973, p. 126), says about this. "In practice, the development of a program can never be performed either in a strictly top-down or a pure-bottom-up direction. In general, however, the top-down approach is dominant, when a new algorithm is conceived. . ." But if one studies programs which are developed in this fashion, there appears to be yet another operation, which I shall call *amendment*, where one modifies code *within* one level of the hierarchy.

Consider the following example: we are designing a program that does a certain numerical computation repeatedly for a certain set of values. We have therefore decided to make a loop whose body performs the computation for one value. At the present level of decomposition, we have specified "do the computation" as one operation in the loop, but we have not written out the details of that computation.

We now decide to handle an additional requirement on the program, namely that the sum of the results from all the computations is to be obtained. We therefore decide on a variable to hold the accumulated sum, we declare it at the beginning of the program or block, we initialize it to zero before the loop, update it within the loop, and finally use the sum (for example print it out, or send it to the next computation), after the end of the loop. All of these are of course abstract steps, which may have to be further decomposed. These changes in the program to achieve one single goal, together constitute one amendment.

Other examples of alternating decomposition and amendment (in this sense) are easily found. For example, example 15.2 in Wirth's *Systematic Programming* (p. 133) combines these two operations. The shift from version 3 to version 4 is one example of amendment rather than decomposition.

The purpose of amendments is not to correct errors that have been committed earlier in the design process, but instead to satisfy one additional requirement on the program (as in the summation example), or to improve the efficiency of the program (as in the quoted example by Wirth). Each amendment is conceptually one single thing that one wants to *do*, but it may result in changes in several different places in the program. It is desirable that amendments are done at the "right" time in the decomposition process: if they are attempted too early, it may be impossible to do them, or one is tempted to put the inserts in the wrong places; if they are attempted too late, it may be hard to see where the inserts are best located.

Consider now the obvious environment for this programming process, where the programmer sits at a console and specifies first the top-level structure, and then the successive decompositions and amendments to the programming system. The simplest implementation is to consider this as a case of text editing, and actually perform the substitutions during decomposition, and the inserts

during amendments. In this case the history of the program development is lost. There are however advantages to the alternative scheme where the system retains the development history, i.e., during decomposition it retains both the name of each step in the algorithm, and its decomposition; and when one amendment causes several inserts into the program, it retains the amendment as a separate entity, with pointers between it and the places in the program where the inserts are to go.

In the ideal decomposition case, where several consecutive steps in the algorithm are decomposed independently and in arbitrary order, this scheme allows one to do each decomposition in the textual framework of the surrounding higher-level steps, but without the tedious details of the decompositions of other steps on the same level. Also, this scheme allows printouts or other presentations of the algorithm where different branches in the hierarchy are represented to different depth.

In the simple amendment case (without regard to decomposition) one advantage with this scheme is for modification. If we later want to remove an amendment in the program, all the data that pertain to that amendment are referenced from one single place. Another advantage is for presentation. Suppose a program has been developed in this fashion, and there are a considerable number of inserts into the skeleton. It then becomes possible to just print out the skeleton and one or two sets of inserts, in order to focus one's attention on them.

A further advantage is for defaulting. If we have a "node" or conceptual entity in the system for this summation operation, then that node should clearly contain a reference to the general concept of summation. The general concept may then provide default information about which inserts are necessary, and what their form is likely to be—for example, that the summation variable is usually initialized to zero.

Finally, for cases of combined decomposition and amendment, this scheme allows one to perform the amendment on the appropriate decomposition level of the program, so one does not have to see the lower levels of decomposition that may be irrelevant to the amendment.

The essence of the argument is that it is sometimes useful to consider a program as a combination of a "skeleton" which has been obtained by successive decomposition, and a set of "amendments". Each amendment is thought about by the user as "one thing the program has to do for the user" or "one trick done by the program", but from the point of view of the actual program, an amendment is a set of inserts into the skeleton, each insert being a statement or "step" in the final program. We use the term insertive program for a program with that structure.

Unlike data-driven programming and advising, the proposed method of insertive programming does not seem to be in current use. It is also not trivial to start experimenting with it, since it is hard to administrate an insertive program without the support of a suitable programming system. By consequence, it is also hard to evaluate the method except by asserting its intuitive appeal. But the method is "in the air" in the sense that some current work on automatic programming (see e.g., Rich and Shrobe, 1975) analyze a user-written program and extract this kind of structure. The proposal here represents a change of approach since I think the skeleton/insert structure can best be obtained as a side-effect of interaction with the user, but the internal representation may be partly similar in both cases.

The idea to "factor out" amendments and represent them outside the main program structure, rather than immerse them in the program, immediately generalizes to a number of other situations. A trivial example is for declarations, which should be associated with a block as an entity, and not be thought of as textually located at the beginning of the block.

Data flow between statements. Still another, and less trivial candidate for factoring-out is for data flow. Consider again an example from the bibliography system. When the data entry program prompts a user for the description of a document, it performs basically the following operations:

- generate an internal name for the document (a "gensym" atom)
- prompt the user for the properties of that document (which involves a loop over its desired properties)
- perform cross-referencing (involving inversion of some of the properties, i.e., the creation of back-pointers from the property to the document name, and also some other construction of new properties)
- save the description of the document on the back-up file

The most natural description of the successive steps have that form, saying "do A, and do B, and do C, . . .". When the steps are implemented in the program, one important thing has to be added, namely provisions for data flow between the steps. In the present program, the atom that is generated in the first step is provided as an argument to the following steps. The second step attaches properties to that atom, and the third and fourth step use those properties.

This data flow can be realized as a program in several different ways. One method is

```
v := generate_name()
prompt(v)
cross_index(v)
save_backup(v)
```

Another method, which assumes that the middle two procedures return their argument as their value, is

```
save_backup(cross_index(prompt(generate_name())))
```

In more complex cases one may want to let one procedure send several values to one or more other procedures, which must be accomplished by packing

several values into a list, or by assigning some or all of them to relatively global variables. Different methods have one thing in common: they clobber up the nice and understandable structure that one had before. Therefore it is here again natural to factor out the data-flow information from the program itself, and specify it in a separate place. At least in routine cases, the programming system should be able to take the responsibility for choosing the appropriate realization of the flow in terms of procedure calls, auxiliary variables, and so on.

The two program fragments above contain only an incomplete description of the real data flow that takes place, since they do not explain that certain properties are assigned (i.e., facts are stored in the data base) in one statement, and accessed in later statements. If the data flow is anyway explicitly represented by statements outside the program, then it would be desirable to describe such data flow via the data base as well (at least) for the purpose of documentation.

Terry Winograd has suggested one more reason for separate specification of data flow (in a private discussion): many common algorithms potentially provide several outputs, but often only some of them are needed. An algorithm to compute the standard deviation may also provide the mean. An iterative computation may return both a result and an estimate of its error. A hashing algorithm may return both the desired result, and an indication that it is time to extend the hash table. The integer "divide" operation returns both a remainder and a quotient. In such cases, one would like to write a general call to the algorithm, and to specify through separate data flow statements which of the outputs are desired, and where they are to be sent.

Direction: conceptual program structure

The original view of a program (for example in machine language) is a coherent document or text whose structure directly reflects the order in which it is executed. Declarations and procedures relax that structure. The methods and ideas described here continue the trend towards the distintegration of that program structure. In data-driven programs, pieces of code are associated with data items that may be part of the description of the data structure (type names and property names in the bibliography system), or of a formalized description of the problem (the PCDB example). In advising, the program is organized as a number of modules which the user may group in any way he pleases. In the proposed insertive programs, major parts of the program body are primarily attached to task-oriented or data-oriented entities.

The logical extension of this trend has been characterized by Charles Rich at M.I.T. as a "raisin-in-the-cake" system. The executable program consists of a number of fragments (the raisins) which are embedded in a structure of declarative or problem-descriptive information (the cake). A part of the system's knowledge is represented through the code in the "raisins", but a considerable part is also represented as the "cake", the threads that lead to a piece of code and cause the system to execute it at appropriate times.

Terry Winograd (Winograd, 1974), in the fifth of his "lectures on artificial intelligence", describes a similar ideal, and uses the term *conceptual program-*

ming. This is a well chosen term: it suggests both that the program is organized around concepts used in a model of the application, and that pieces of coherent code are represented in terms of underlying program concepts (such as the proposed inserts in insertive programming), rather than the conventional surface-structure program. I have therefore used the same term for the title of this paper.

Conceptual programs might not be appropriate for all purposes. If one legislates that important programs must be checked and authorized by a "data ombudsman" or a board of data processing, for example to safeguard the privacy of data, then one certainly wants a closed form of a program which can be inspected, authorized and locked up in a safe for reference purposes. The representation of the program as a document will then be the most appropriate for the foreseeable future. Conceptual programming would probably be used first in experimental-programming and pilot-system situations, where a program is developed in order to improve the understanding of a problem, rather than for actually being used.

In one particular case it is already common practice to use several different "projections" of the same program structure, namely when we use cross-reference list generators. The situation is then that we have a representation of the program (namely the listing) in which some items which we want to see together (namely all occurrences of the same symbol) do not stand close together. The purpose of the cross-reference generator is to transfer the program to another projection, where the logical proximity is physically apparent. The point with conceptual programming is that one should instead attempt to create the underlying conceptual structure, of which both the conventional listing and the cross-reference table are projections (Fig. 7).

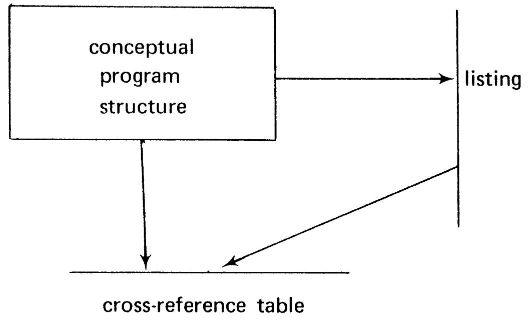

FIG. 7

PART III

A STUDY OF THE BIBLIOGRAPHY SYSTEM

"Conceptual programming" as described in the preceding chapter is an ideal

rather than a method: it represents a plausible direction of research, but not a definite method that can be tried and evaluated. One purpose of the present study was to make the proposal more specific, with regard to the programming methodology and also the supporting programming system.

One way of achieving that purpose might have been to start building a system and see where it would lead, in the tradition of experimental programming. That method was rejected since the resulting system could easily become overloaded with features which were introduced on grounds of generality, aesthetics, or accidence, but which were not really needed. I wanted to know which facilities would really be useful. Two alternative methods were, either to do paper experiments with program development while *pretending* to have a suitable programming system, or to take an actual program, already written by someone else, and try to learn something from it.

Of those two alternatives I preferred the latter, mostly because it offered an opportunity to work with a program of realistic size. A paper experiment could probably not be pushed beyond the level of a toy program of one or a few pages, and I am not convinced that what one can learn from such small programs has an application on programs of more realistic size and complexity. Also, if a paper experiment was to be conducted, it seemed appropriate to first work with an actual program, and then redevelop a program for the same purpose, using the proposed techniques, as the paper experiment.

Dave McDonald's bibliography system, which was described in the last section, was chosen as the object of study. It is highly parametrized and uses data-driven procedures in several ways. and I believe that data-driven-ness is a good first step in the direction of conceptual programming. The present section reports on the observations that were made in the program, and the suggestions that the program implied for how the simple idea of data-driven-ness can be extended, and how a conceptual programming system should be designed and used.

Although the material in this section orginates from the study of one concrete program, it is not intended as an empirical study. Observations made on one single program can of course not really be used as empirical evidence for anything. It is rather a set of suggestions that are claimed to have not merely intuitive appeal, but also a certain concreteness and applicability derived from a contact with a practical program.

Most of the phenomena that will here be described as "observations made in the bibliography program" are fairly standard programming techniques in the LISP-based artificial intelligence community. I believe that roughly the same observations could have been made in another program, or in a program for the same purpose written by another person. However, usually these techniques are implicit: they are used, but never explicitly described or discussed. It was only when I forced myself to study one particular program in depth, that I was able to articulate the methods that are actually being used. When the present paper repeatedly talks about "observations made in the study of the bibliography program", one should realize that these observations did not involve much surprise.

The order of presentation of the observations and conclusions will be mostly historical, particularly since some observations build on others. As a consequence there is at the time a progression from the fairly obvious to the more significant. Experienced LISP programmers may want to skim lightly over the next few pages and start reading more carefully at the data-structure model (page 248).

The first step was to study the dispatching mechanisms that the bibliography system used, i.e., the chains of references which would lead a calling program through data items to procedures to be executed.

In some cases, the dispatching mechanism was simple and straight-forward, and fit perfectly to the preconceptions presented in Part II. For example, each type name (for types of documents) is associated with a procedure which is executed each time a document description of that type has been entered.

Aggregates

In two interesting cases, the driving data had a non-trivial structure, and seemed to require a *dual description*, namely both as a program in a specialized programming language (in the sense discussed on page 236), and as an aggregated structure that contains a number of data entries. That dual description seemed like it would be a concrete problem for a cônceptual programming system.

Let us first describe an example of this, which appears in the printout program, that is the program that produces a nice-looking printout of the properties of a sequence of documents (for use as the "references" section at the end of a paper). The choice of which properties to print out is determined by the type of document (book, article in journal, etc.), and the format is to be variable to allow for different conventions. The bibliography system does that as follows: each set of conventions is assigned a name, which of course is an atom, and which is called a *recipe-name*. The recipe-name has a property under the property-name OUTPUT-RECIPE; whose structure is

```
((pn  params  pnct)
 (pn  params pnct)
  . . .
 (pn  params  pnct)
 ((type  (pn params pnct) (pn params pnct) . . . )
  (type  (pn params pnct)  (pn params pnct) . . . )
  . . . ))
```

Here each *pn* is a property-name for document id's, for example AUTHOR and TITLE; *params* is a sequence of flags which signal specific conventions (for example whether the authors' first names go before or after the last names); *pnct* specifies how to terminate the current property (with a comma, full stop, carriage return, etc.), and *type* is a document sub-type such as BOOK. The printout

program interprets output-recipes as follows: it scans the top-level list, and for each triple of the form <pn params pnct>, it looks up and executes a procedure stored as get(pn, PRINT-UP-FN). That procedure has access to the current document id (so that it can look up its property under the property-name *pn*), and to *params* and *pnct* which specify some details of its operation. When the scan arrives to the sub-lists marked with type names, it chooses the sub-list marked by the type of the present document, and proceeds to scan down its list of triples, but ignores the other branches. Thus the end of the structure serves as an implicit **case** statement.

There are two ways of thinking about such output recipes, namely as a program or as a location structure (a structure that contains locations that the user can put things in). First, consider the person who has received the instructions for authors in a certain journal, and wants to encode them for the bibliography system in its present operating environment (source code on text files, etc.). That user will want to write up one recipe for that journal, and will be inclined to think of the recipe as a program, which in each step "does" a printout of one item in the document description. For example, he would be inclined to welcome a facility for insertion of arbitrary LISP code in the middle of a recipe which enables him to do "what he wants".

On the other hand, suppose the bibliography system is embedded in a conceptual programming system which "understands" some of its structure, and suppose the user has just told the system that he wants documents to have one more property, namely an NTIS number. The system should then lead the user by the hand and ask him for additional information that the system needs to know, for example how such numbers are to be prompted, and printed out. The system must therefore have a model that enables it to tell what information has to be added where.

It seems to me that such a model could best start with saying that the actual data structure implements a number of mappings, namely

PROPS: type-name → sequence of property-names
TOPRINT: recipe-name * property-name → <params, interpunction>
PRINTFN: property-name → procedure

It should also know how those mappings are implemented: the last one is stored in the trivial way as a property; the first two are stored together in a somewhat non-trivial manner in the output-recipes. But the mappings would be useful as a backbone structure to which one can attach additional information e.g., about how the mappings interrelate, and the first mapping could be shared with other parts of the system, since it is really an ordinary declaration.

This was one of two examples in the bibliography system, and an additional very similar example is provided by the file description mechanism in the INTERLISP system. Let us use the term *aggregate* for a structure which one sometimes wants to treat as a program in a specialized sub-language, and sometimes as a data structure that contains a number of data items which the user will want to address and modify individually.

The problem that aggregates pose for a conceptual programming system is that both views of the aggregate must be "sugared" or explained to the user, and the system must therefore be able to move freely between the two views. If the user is to think of the aggregate as a program (for example to input an input-recipe), then the syntax and semantics of the program must be explained to him. If he wants to think of it as a composite data structure, then he wants to communicate with the system in terms of underlying structures such as the mappings shown above. If the user is working with one representation and wants to change it, the system should be able to implement that change in the user interface to the other projection, or at least ask the user the right questions about it. This means in particular that the system must know for each projection both how to talk to the user in its terms, and to understand the user when he uses the terms of that projection.

With the data-structure view of aggregates, one would hypothesize a "pure" representation which more directly corresponds to the mappings, and an "aggregation" operation where the composite structure is constructed. At least two reasons for aggregation can be seen in the bibliography program: execution speed (if one knows that the contents of a number of locations will be inspected together, then it makes sense to form a composite structure such as the output-recipes and store them all in there), and presentation for the user (who sometimes may want to see them together, and therefore may be inclined to group them together on the listing of the system).

Another example of aggregation. The bibliography system contains one more example of aggregation, which will be described shortly for the completeness of the record. The data entry program (which prompts the user for the properties of successive documents) assumes that each type name (for sub-types of documents, for example BOOK) is associated with a list of triples of the form

<<prompt-word, explanatory-phrase, property-name> . . . >

In prompting the user for one document, it first asks about its type and looks up the corresponding list, and then for each triple in the list it prints out the prompt-word, uses the explanatory-phrase if the user requests *help*, stores the input as the property of the document associated with the property-name, and also executes a data entry procedure which may be associated with the property-name. The following abstract underlying structure is natural:

PROPS: type-name → sequence of property-names
TOREAD: property-name → <procedure, prompt-word, explanatory-phrase>

Again the aggregation chosen in the bibliography system is convenient when the user reads and edits a text file of the program, and allows fast execution, but a conceptual programming system should also know the underlying structure.

Dynamic modification of aggregates; self-modifying programs

A second observation was the usefulness of dynamic modification of aggregates. If the aggregate is viewed as a data structure that one makes a loop over, then this is just dynamic looping, where the body of a **for**-loop changes the value of the loop counter. If the aggregate is viewed as a program, it instead amounts to having self-modifying programs. The latter view is interesting considering the present apparently universal belief that self-modifying programs represented a primitive stage in the development of machine language, and that they are antithetical to well structured programs. The observations here provide a counter-example.

Consider the feature whereby the reading program may default some properties. The properties for "reports" (technical reports, internal memoranda) include INSTITUTE (where issued) and LOCATION (the city where the document was published, which usually is the city where the issuing institute is located). Sometimes the system may already know the location of the institute, and it then does not have to prompt the user for it. Several similar defaults are possible. Such defaulting is handled by the bibliography system as follows: each time a document description is entered, a fresh copy of the "prompt-list" aggregate is created. It may contain

< . . . <INSTITUTE . . . >, <LOCATION . . .> . . . >

A counter steps down that list as successive properties are prompted, and executes the procedures associated with property-names as described above. The procedure associated with the INSTITUTE property tries to default the LOCATION property, and if it is successful, it deletes the triple that would ask for LOCATION, from the current aggregate. (If that is the next triple, it can simply step the counter, otherwise it has to edit the current aggregate). The same method is also used for choosing the right prompt-list for each type: the fresh copy of the aggregate that each document obtains initially starts as

<<TYPE . . . >, . . . >

and only contains those properties which are common for all types. Thus the first prompt asks for the type of the current document, and the procedure associated with the TYPE prompt adds the specific prompts for the current type, to the current aggregate.

This method is not just a hack! The initial "program" is very readable, since it just says what the system is to ask the user about, and the program modification operations are also very easy to understand, since they can say almost literally, e.g., "remove the question about the LOCATION property". They manage with existing primitives, and do not need to introduce additional symbols or constructs. By comparison, if the same defaulting process is to be performed by a conventional "structured" program, one would have to have a loop over the possible properties, plus a mechanism with boolean flags or other similar devices in order to remember from one cycle to the following which properties are to be suppressed. In the reverse case where one sometimes wants to *add* prompts chosen from a large number of possible properties, or modify later prompt-words, or the order of the prompts, it is almost impossible to avoid making use of a structure which in some sense is a program, and which is dynamically modified.

Dynamic modification of aggregates adds one more criterium for aggregation: besides improving readability (by clustering the right things together) and improving execution speed, an aggregate of properties which may be used as a self-modifying program, should be designed to contain those things which will change dynamically. For example, if one wants the result of one interaction to be able to modify the prompt-words for subsequent interactions (Hägglund, personal communication, has described a case where that was desirable), then it is necessary to locate promptwords directly in the aggregate, rather than store them as properties of items that appear in the aggregate.

Criteria for the model of the program

Conceptual programming is characterized by the disintegration of the conventional programstructure. The obvious question is then "what structure comes instead". That question was sometimes acute when studying the listings of the bibliography system. When data-driven procedures were involved, one often had to work hard to understand what goes on, and there were also places in the program where a data-driven call would have been natural but had actually been avoided, probably because it would have scattered the information too much (in the listing and/or for the execution of the program). The **case** construct in the cross-indexing program (described below) is an example.

In the conceptual programming system there should therefore be a structure which keeps track of the miscellaneous data-driven procedures, and is able to explain their location and their purpose to the user. That structure will be a model of the entire system from one particular viewpoint, namely the data-driven calls. There are four major requirements on the model: it should be sufficiently precise that it can be stored in the programming system; sufficiently palatable that the user can read it and enjoy it; sufficient, i.e., contain enough information for its purpose; and minimal, i.e., it should not contain things that it and the user do not need to know. I attempted to extract such a model from the listings of the given bibliography system.

Second-order data-structure models

The first hypothesis was that a second-order data-structure model would be appropriate. The model can best be explained as a generalization of ordinary declarations (in languages which have such). Ordinary declarations for the bibliography systems might say for example "objects of the type JOURNAL shall have the following properties:

property-name	structure of property	purpose
FIRST_YEAR	integer	year of issue of first volume in present series of the journal
SOCIETY	atom	name of the organization which issues the journal
LANGUAGES	list of atoms	official languages of the journal

Such information may be specified formally in a declaration, or informally to the user, and it is sufficiently palatable to be appropriate for the conceptual structure. But if that information is stored in a data base, there must also be specifications about how it is to be stored. What has just been given is then a first-order data-structure model, and one needs a second-order model which says things like: "all objects shall have a *type*. Type names are represented as atoms with the following properties:

property-name	structure of property	purpose
ISA	other type name	type of which this is a specialization
PROPS	list of atoms that are used as property -names	properties that objects of this type may carry

Each object may have properties, with the names specified by the PROPS property of the type of the object. Property-names are represented as atoms with the following properties:

property-name	structure of property	purpose
PROPSTRUC	one of the atoms: INTEGER ATOM LIST-OF-ATOMS	specify the allowable structure of properties with this name

For example, if JOURNAL is a type-name, one may have:

get(JOURNAL,PROPS) = <FIRST-YEAR,SOCIETY,LANGUAGES, . . . >
get(FIRST-YEAR, PROPSTRUC) = INTEGER
get(LANGUAGES, PROPSTRUC) = LIST-OF-ATOMS

The above is only given by way of example, and in actual use one would certainly want to improve on the description. For example "type-name" should itself be made into a type, and so should "structure descriptor" (ATOM, INTEGER, etc.). Also, ISA links must sometimes be handled in a less simple-minded way. The second-order description clearly allows a formal representation. When presented to a human, it tends to be less palatable than the first-order model because it is more abstract, but that may be compensated with the use of examples.

The conjecture was now that data-driven procedures could be explained to the user by making them part of the data-structure model. The second-order model would then say that there are such things as *types, property-names,* and *structure descriptors.* Among the properties of type names one would find PROPS and as given above, but also a CORELATION-FN property which is a procedure, executed each time a document of that type has been entered. Among the properties of property-names, besides PROPSTRUC above, one may find a READFN property, executed each time a property with that name has been prompted for, a PRINT-UP-FN property used by the printout program, etc. For aggregates, the second-order model would account both for the idealized underlying data structure, and its actual implementation.

Some of the data-driven procedures in the system could actually be accounted for even in a first-order model, namely procedures associated with data items which may appear in input, such as abbreviations. Most of them however required the second-order model.

An immediate observation was that data-driven procedures should not be modeled alone, but together with parametric information which the user could communicate to the system by storing it in the data base, for example prompt words, lists of properties to be written out on files, etc. The distinction between data-driven procedures and parameters is vague anyway, and more important, parameters appeared in the access paths to procedures, for example in the above-mentioned aggregates. The purpose of the model should therefore be characterized as *describing the user-filled locations* in the system, where a *location* is a place which the user can characterize by an atom or combination of atoms, and fill with a parameter or a procedure.

Work on the model was carried fairly far but was not completed. On informal evaluation, the model seemed to satisfy the minimality and the precision requirements fairly well, but it was hard to make it sufficient and palatable. The reason seemed to be that it could not live alone. It is clear that the data-structure model must contain at least informal references to the program structure: in describing the purpose of a data-driven procedure, one would specify which program or procedure it is called by. But it turned out that short informal references were not sufficient; one really had to say fairly much about the program structure expecially in order to explain how to write procedures to put into the data driven-locations.

Program-structure models

The next step was therefore to also develop a program-structure model, which should describe the programs and program parts in the system with respect to their means of communication (data flow, direct calls, and data-driven calls). The top level of the model was a data flow model similar to Figure 6 although somewhat more detailed. Each "program" compartment was then subdivided

into procedures with a procedure-call structure.

Predictably enough, this model required some references to a data-structure model, but it turned out that fairly few references were necessary. When written out as a natural-language text (i.e., candidate program documentation) it specified first the top-level data flow model, then the first-order data-structure model in the major "data pools", and then described each constituent program box. The resulting description turned out to be a good framework for describing the user-filled locations. Essentially each location was described in the context of the program which used it. Only one location was used by more than one program, namely the TYPE property of documents, and it was documented in the introductory data-structure chapter.

Data-structure-based programming

A possible interpretation of these two modelling attempts might be that the program-structure model is in general the "natural" and "primary" one. There is however another possible interpretation, namely that the system *had been written* with a program-structure model literally in the mind of the programmer, and that that is the reason why the formal program-structure model seemed to fit the system better. The discussions with McDonald confirms this: he had thought of the system as a number of programs *with handles on them*, which would enable the skilled user to modify the behaviour of each program, i.e, his model of the system was mostly a program structure.

The data-structure model of the system might then be more appropriate if the system had been designed with a data-structure model in mind. Concretely, one would start out by designing the second-order data-structure model, and the top level of the program-structure model. One would then organize the programs around the data-structure model by working downwards to write data-driven procedures that should be associated with the model, and "upwards" to write the procedures that use the data-structure model and call the data-driven procedures. I shall refer to this method as data-structure-based programming.

Evidence that gives tentative support to data-structure-based programming

There are in fact some designs in the bibliography system that probably would have been done differently and better if that method had been used. On balance, other things might of course have come out worse then, but it is still of some interest to discuss the designs that point in the direction of data-structure-based system development.

Consider the following example. The saving program (for transferring data from the "active" to the "passive" data base) makes a loop over all the properties of each document to be saved, and prints them out. Each property-name, for example AUTHOR, has a dumping procedure associated with it. (These procedures are designed to improve the appearance of the files, in order to facilitate text editing on them). However, since several property-names may use the same procedure, an extra level of indirectness is inserted: the property-name is associated with an atom, which in its turn has a dumping procedure, and several property-names may share the same such atom.

This common-sense arrangement is easily explained in a program-structure model. But it is interesting to see the names of the intermediate atoms. They are: PNATOM (a term used in the system to characterize an atom in which the distinction between capital and small letters is significant), LIST-OF-PNATOMS, TEXT, and CODE. In other words, one dumping procedure is needed for properties whose structure is to be a list (denoting a set) of atoms, another for properties which are procedures etc. The names of the procedure indicates the structure of the property.

If the second-level data structure model had been developed first, then such structure describing entities would already exist and be ready for attaching procedures to. That design method would then encourage economy of concepts through multiple usage of the same symbol, since the data-structure model is useful in itself (for description and documentation of the system), and also since several different procedures may sometimes be attached to the same symbol.

Another example. The bibliography system offers additional examples that point in the same direction. Here is one: for the purpose of the same saving program, the system maintains a catalogue for each type of document. Since BOOK is a document subtype, there is a global variable BOOKS whose value is a linear list of all document id's of subtype BOOK. That linear list is updated as follows: the data entry program prompts the user for all properties of the current document, and then executes the procedure get(t,CORELATION-FN), where *t* is the type of the document (for example, BOOK). That procedure adds the current document to the appropriate global list. Thus get(BOOK, CORELATION-FN) is

```
(LAMBDA () (ADDL 'BOOKS REF))
```

where *ref* is a free variable bound to the current document id, and *addl* does the right thing. Thus again the catalogue names are used only as arbitrary global variables, but they happen to contain information that duplicates the information in the type name. In a data-structure-based system development, one would exploit that correspondence more systematically by making it part of the second-order data-structure model that each subtype of documents shall have a catalogue as its (say) CATALOGUE property. (The property might be the actual list of document id's or the global variable whose value is the list of document id's). Such a decision would make the system more self-documenting. In reading the bibliography program, non-trivial effort was required to find out that there is a correspondence between type names and catalogues, as given by the following table:

type name	catalogue name
BOOK	BOOKS
THESIS	THESISES
REPORT	MEMOS
JOURNAL*	JOURNAL-ARTICLES
COLLECTED	COLLECTED-ARTICLES

*) meaning a paper which was published in a journal.

Finally, with a data-structure-oriented design, one could do away with the CORELATION-FN handle, since its only use in the present system is to increment the catalogue, i.e., to contain the knowledge of the above table.

Dynamic use of locations. The latest mentioned example, about catalogues for document types, also suggests another observation. Our description of "data-driven programming" and the "raisin-in-the-cake" approach in the previous section centered around the *procedure* as the essential component of the system. In forming the data-structure model, it became at once necessary to also include parameters in the model. The latest example indicates that one should also include global variables, or in general, storage locations whose contents are modified during the execution of the program, and which are accessed from such widely different parts of the program that they are considered as global. Therefore, the updated description of data-structure-based system design is that one would start out with a (usually second-order) description of the desired data structure, and when in the course of program development one needs a location to store something in, be it a procedure, a parameter, or global intermediate data, one attempts to choose the name of the location as an entity or a combination of entities that already exist in the data-structure model.

An obvious question is obtained by extrapolation: what about *local* variables or storage locations? There are several obvious reasons for not storing local data in the global data base, and it seems that in most cases one would not gain much with it. If anything, one might let the conceptual system contain definitions of the purpose of some local variables in terms of the global data structure, but other definitions of local variables (in terms of local program structure, data flow, or purposes of code) may often be more useful.

A possible objection to data-structure-based system development is that the suggested ordering is not feasible: one cannot in general define the whole data-structure model first, before one even starts thinking about the program. In our latest example, one might not realize that the catalogue is needed until while one is writing the program. The observation is correct, and the proposal is certainly not that the data-structure model should be designed *and frozen* before the actual programming starts. The proposal is instead that one tentative data-structure model shall be designed at an early stage, and that the program shall be designed "around" it (in the sense described above), but it is evident that if in the course of writing the program one wants to modify the data-structure model, in order to satisfy the needs of the program, he should be able to do so. One obtains the right perspective by thinking of a first-order model as equivalent to the declarations at the beginning of a program in a programming language that uses such, and of a second-order model as a generalization thereof. One tends to write declarations before the body of the program, but one also feels free to modify them when needed.

A third example. The bibliography system did not of course provide any conclusive evidence for or against the utility of data-structure oriented system development. Separate experiments on that issue seem worthwhile, and can be performed without first developing an advanced conceptual-style programming system. The system did provide one additional example where a data-structure model would have been useful, and that example shall be shortly reviewed here for the completeness of the record, although it does not enable any additional observations. One part of the data entry program has as its major purpose to do "cross-referencing" or property inversion. Thus if the J-NAME property (for journal name, namely the journal in which the paper was published) of the entered document was JACM, then the MEMBERS property of JACM should be augmented with the current document id. That operation is implemented by dispatching on the property name (actually through a construct similar to a **case** statement, rather than through an actual data-driven procedure call, but that is not essential here), and looks as follows for almost all property-names:

```
(COND . . .
    ((EQ TAG 'J-NAME)
     (ADDL 'JOURNALS VALUE)
     (BACKREF))
 . . .)
```

or in an Algol-like notation:

```
. . .  elseif tag = J-NAME
       then begin
            addl(JOURNALS, value);
            backref()
            end . . .
```

Here the procedure *backref* performs the property inversion as just described. But the block also serves a second purpose, namely the call to *addl* which adds the current property value (in this case, JACM) to the global value of the variable JOURNALS, if it was not already a member. Thus an atom which appears as a property under the property-name J-NAME is implicitly defined to have the type journal (implying that it can have certain properties), and should be included in its catalogue JOURNALS. Again there is a correspondence table of the form

property-name	catalogue name
J-NAME	JOURNALS
AUTHOR	AUTHORS
UNIVERSITY	UNIVERSITIES
. . .	. . .

which is implicitly represented in the code associated with the property-names, as arguments to *addl*. In a data-structure-based design one would be inclined to maintain the catalogues (or catalogue names) as properties of the property-names in the left-hand column. Furthermore, in the actual system the accumu-

lated catalogues are used by a call in the saving program of the form

```
(FREEZE-DRY '(AUTHORS TITLES JOURNALS . . . ))
```

It would be natural anyway to maintain that argument list in a global location, which then in a data-structure-based design would be just the union of the sets of property-names for all document types. Finally, the procedure *freeze-dry* is actually parametrized with respect to a DUMPPROPS property of catalogues such as JOURNALS; that property is the list of property-names for properties that are to be saved. That item again is a natural part of the second-order data-structure model.

Description of program generation in the program-structure model

Switching back to the program-structure model, it became necessary to describe there not only communication between programs (by data flow and through invocation), but also generation of programs. This was necessary both for the obvious reason that the system assists in the generation of some of its data-driven procedures, but also in order to describe the dynamics of generating and re-loading data files. These points will now be discussed in more detail.

In a conventional data-flow model, one would have two kinds of nodes or boxes, one for "programs" and one for "data". That distinction was not appropriate in the present case, for two reasons. First, there were instances of low-key program generation, where one program would take input from the user and generate appropriate, executable code, which would be called from other parts of the system later during the run. Instead of the program/data dichotomy, it was appropriate to have one single kind of box, but two kinds of arrows that go to a box: an "input" arrow and a "create" arrow. If program box P uses data from box D as input, then there is an arrow from D to P with an "input" attachment to P. If P then generates data in the box D′, there is an arrow from P to D′ with a "create" attachment to D′. In the program-generation case, one box may have both an incoming "create" arrow, incoming "input" arrows, and of course both types of outgoing arrows.

A philosophical problem arises in the representation of procedure calls, for example when generated procedures are called from elsewhere. One would like to consider procedures as special cases of data, and use the same notation both when a program calls a (possibly data-driven) procedure, and when it uses more declarative parametric information provided by the user, particularly since there is a continuum of ways in which the program may "interpret" those parameters. One approach is to define an "inspect" arrow which is used both for procedure calls and for inspection of parameters, and which runs from the caller to the callee, or from a procedure to the parameters it inspects. But then the latter case may be considered as a special case of "input", and could equally well be represented by a data-flow arrow with an "input" attachment—except that the latter was supposed to point in the other direction. One may therefore decide to have just one kind of arrow, for data-flow, and to represent procedure calls as data flow from the callee to the caller, or one may attempt to distinguish between object-level data flow and inspection of parameters. The former convention is cleaner but takes a while to get used to.

There was also another reason why the program/data dichotomy was inappropriate for the modules in the data-flow model, namely that it was often natural to clump together some kinds of declarative information ("parameters", "data") and some groups of procedures into one entity, and consider the whole module as "parametric". The fact that different parts of the information contained therein related somewhat differently to the LISP interpreter (or compiler) was very irrelevant for the logical clustering.

Files as a special case of programs

The bibliography system also used a less obvious kind of "program generation", namely the generation and subsequent loading of files. Here an explanatory detour is necessary for the reader who is not used to incremental languages. In a conventional language, it is commonplace to have one program which prints out data on a file, for example the current values of some variables, and a corresponding program which reads the file and recreates the same state. The programs must be coordinated: if the printing program produces variable values in a certain order, then the reading program must set them to input data in the same order. In an incremental programming system which performs a read-evaluate-print loop, one usually prefers to let the printing program generate a series of expressions on the file, in such a fashion that the file can be read by the top-level read-evaluate-print loop, and then produces the desired result. Thus if the current value of the variable A is 3, the printing program will print out

A := 3

or whichever representation of that assignment is used in the programming language. The advantage with that design is that no special reading program is needed, and therefore most of the coordination problem goes away, making it easier to modify the printing program. Also, flexibility can be achieved without loss of transparency. In the conventional print-read pair of programs, one soon gets to the point where the printing program must print out signals or flags which tell the reading program how to handle the subsequent data. Such signals must then be decoded by the reading program, and continued coordination during program changes requires that they must be documented. In the incremental programming system, groups of data are never made to appear "free", but only as arguments to procedures which know what to do with them. If at some time the printing program is extended so that it will print out additional expressions that call a not previously used procedure, then no other code need to be affected.

In a conventional programming environment, with strict distinction between "programs" and "data", an output file should be represented in the program-structure model as a "data" module, which may be the output of one program

and the input of some others. But when the file consists of a sequence of expressions which are to be evaluated, and which contain calls to procedures defined in other program modules then one must consider the file producing program as a program generator, and the program that causes the file to be read, as doing a call to that file-program. The latter observation also gives additional support to the idea of considering any procedure call as a data flow from the callee to the caller.

Summary of program-structure model

The program-structure model of the system was not completely finished, but it proceeded sufficiently far that I was convinced I had seen most of the interesting problems. The design of the model was the obvious one, given what has been said above, and can be summarized as follows: on the top level, there is a data-flow structure between a number of blocks or modules. Each module may contain procedure definitions, data, or both. Arrows represent flow of object-level data, but also generation of and calls to procedures (or groups of procedures). For data-driven procedures, all procedures stored in the same location (for example all procedures stored as getp(p, READFN) for some property-name p) would be in one block, and data-driven calls to them are described like any other calls.

The modules in the top-level model are crudely divided into three groups:

- **program modules**, which could be explained to the user-reader as for example "this is the program that inputs contributions to the data base by prompting the user";
- **parametric modules**, which contain parametric data and data-driven procedures;
- **object-data modules**, which contain actual descriptions of documents in one or another representation (for example property-lists, or as a text file).

As assumed at the outset, the purpose of the program model was to explain not only what the system "does", but also how the user can change the contents of the parametric modules in order to modify the system's behaviour. It therefore referred to a declaration-like description of the object-data modules, and then for each program module described what handles are on it, and how they are used.

In a more exhaustive description of the program, one would also document each procedure in at least the program blocks, specify its purpose, and describe the calling structure within the module. For the limited purpose of documenting the use of parameters, it was found unnecessary in most cases to make such an analysis; in a few cases a limited breakdown of program modules was useful for explaining when the handles would be accessed, and thereby, for explaining how they should be used.

The need for an initialization model

Both the program-structure model and the data-structure model had to be painstakingly extracted from the listings of the program. The considerable and somewhat unforeseen difficulty of that process raised another question: how do these models relate to the program listing? By what mechanism can one relate items in the description, to items in the listing?

One might have expected this problem to be trivial, on the following grounds: the purpose of the listing is that when it is loaded into the system, it should fill the locations which will be needed when the system is run. Here "location" is used in the sense defined above, i.e., a procedure name, or a property or other place where one item (a parameter, a data-driven procedure) is stored. The listing should therefore consist of a set of expressions, each of which fills one location, and it should be divided into modules that correspond to the modules given in the program-structure model.

Unfortunately things are not that easy. Both in the bibliography system and in many other cases, the programmer organizes his files and the process of assigning locations in a more sophisticated way, for example as follows:

- For some types of parameters and data-driven procedures, one wants to define auxiliary entry procedures, and then use them for input of parameters or procedures. The auxiliary procedures may serve to provide a more compact input (by eliminating repetitions of the same items), and/or to improve the legibility of the text file. When such auxiliary procedures are used, they must of course be input to the system before the expressions that call them.
- Sometimes a group of parameters and/or procedure definitions are separated into their own file, for example in order to variablize over users, or to facilitate restart after certain kinds of crashes, where only some of the locations need to be re-initialized.
- Sometimes locations are not filled when the files are read, but instead the files contain the definition of specific initialization procedures, which are to be called by the user after the files have been loaded. This arrangement is another way to facilitate restart after crashes, but may also be used in order to arrange that all initializations happen in the right order, although they are given in the "wrong" order on the file (in order to make the file easier to work with).
- Sometimes the initialization procedure is defined in one file, and later called from the same or another file, but it may be recalled by the user after a crash.
- Sometimes one location or group of locations may be initialized in several different ways, from several procedures or files.
- Sometimes the main program that the user has to relate to (for example the prompting program) performs its own initializations when it is first called, before it goes into the interactive loop. Sometimes it behaves differently the second time it is called, so the first time it must make a note that it has been called, to be seen the second time; but then if a more thorough restart has been done inbetween (for example by reloading certain files) then that note must be overwritten. And so forth.

In summary, the listings of the system cannot be viewed simply as a set of assignments of contents to locations. A better approximation is to say that the files are together a program whose purpose it is to fill those locations. But even that description is insufficient because of the initializations that take place after the files have been loaded.

Proposal for an initialization model. A sufficient model must start out by separating two questions: first, how is the system loaded, meaning how are locations initialized with their contents, and second, how is the system run, using the contents of the locations. (The distinction may not be aboslute but I believe it is sufficiently clear to be useful). Both questions assume of course a model of what locations there are, i.e., the second-order data-structure model that was proposed earlier. For describing how the system is loaded, one must start with a *description of an interactive session*, for example as

a. load files containing procedures and parameters
b. call initialization procedures
c. call the top level procedure of the system

where the names of the files and procedures must be explicitly stated for each system. Either or both of steps b and c may sometimes be omitted, and additional steps may sometimes be necessary to complete the picture, for example, the execution of the user's entry file which is done when the LISP system is entered.

The description of the interactive session is in fact a program, although it may not be fully specific about the relative order or number of repetitions of certain steps. It may even be data-driven, for example if an initialization procedure wants to load a file, but asks the user for the name of the file, or asks the operating system for the current user id in order to load the correct file. But more important, the session description is similar to declarations in the sense that it is assumed to be relatively constant while the program is debugged and extended. The contents of a file may change, but the name of the file as found in the session description does not need to change except very occasionally.

The purpose of such a session model or loading model wold be to describe how locations are loaded with contents. In order to do that, the loading model must be related to the data-structure model and the program-structure model. The latter relationship is easily done just by extension downwards: the session model contains statements to load a file or call a procedure, and each file and each procedure can again load a file or call a procedure. Thus an ordinary calling-structure diagram is appropriate. Notice however that some parts of the model may be "less variable" than others. For example, assume in some system that the file INIT is to be loaded first, and that INIT shall contain commands to load certain files, so that the system can be extended by adding more file calls to INIT without telling the user. Then the statement that INIT shall be loaded during the interactive session is "less variable" than the loading commands contained in one generation of INIT. The combination of the initialization model and the program-structure model must account for such distinctions.

Relationships between the initialization model and the data-structure model. At a certain level in the program-structure model for initialization, one will find procedure calls or other similar operations which serve to initialize one or a few locations, and where the arguments in the procedure calls are often constants (or "quoted", to use LISP jargon), namely they are the intended contents of the location, or data which are to be transformed in order to generate those contents. (Complications are of course possible, for example when data are first stored in one location and later retrieved, rearranged, and stored in another location.) Let us call these the elementary initialization operations. Above them in the calling structure, one finds procedure calls which mostly serve the purpose of grouping the elementary initialization operations into natural groups; below that level one finds miscellaneous auxiliary or system-level procedures.

Each elementary initialization operation is then characterized in three ways: in which group (procedure or file) does it occur; which procedure is called there; and which location(s) are filled by this operation.

When describing a system, it seems natural at first to maintain the references in that direction, and in particular, to specify for each elementary operation which locations it fills. But it may be advantageous to maintain the system in the opposite way, i.e., to restrict the initialization model to a "skeleton" (in the sense of the term used in the section on advising, page 237) which extends down to but not including the level of elementary initialization operations, and then for each location maintain a reference to the point in the initialization skeleton where an elementary initialization operation for it is to be located. This would be a special case of advising: if the user decides to establish another location in the system, he specifies not only the ordinary declarative information, but also a pointer to the place in the skeleton where the location is to be initialized. This information is retained and may be used both for specific questions ("This location does not seem to have been initialized. Where was that supposed to have been done?") and general questions ("Has initialization of all locations been accounted for?").

A possible objection is that the same purpose may be achieved with simpler means, for example a conventional cross-index of the text file of the system. But maintaining the pointer from the location name to the initializing operation in the system has the advantage that it enables more informative answers. If the user wonders why a certain location has not been initialized, then the listing of the system plus the cross-reference list will at best provide the information whether the listing contains a statement that is intended to do the initialization. A

rogramming system that has access to the reference may instead potentially ive the answer "that location is supposed to be initialized by file FIE, which is oaded by file FOO, but this time FIE has not been loaded because FOO ombed out before it got that far". For another example, suppose the procedure *storesym* has been defined as

```
storesym(a,r,b) = begin
                      addprop(a,r,b);
                      addprop(b, rev(r), a)
                  end
```

and suppose the user has by oversight arranged his files so that storesym(BOY, SUBSET-OF,MALE) is executed before put(SUBSET-OF,REV,SUPERSET-OF) has been performed. When the user later wonders why get(MALE,SUPERSET-OF) is not defined, the system should be able to explain that to him, if it maintains an initialization model and the knowledge where in the model each location is supposed to be initialized. Clearly these are pointers from the second-order data-structure model, to the session description and its expansion into an initialization model.

It may also be remarked that cross-reference listings are not so appropriate when locations are named by combinations of two or more items, which we have argued is a desirable practice. If one has achieved an economy of names, and if the combination of names is significant, then the cross-reference will tend to generate a bigger "fanout" than if single names had been used, and therefore it will assign an immediate relationship to items which may not be at all related.

We assumed above that a distinction could be made between the process whereby the system is loaded and locations filled, and the process wherein the system is run and the contents of the locations are used. One can of course not expect these two aspects of the program to be physically separated in the code: some initialization may be performed when the execution has already progressed far, for example by default the first time the information is needed. The assumption is merely that at a certain level in the program-structure model, one can make a reasonably clear distinction between initialization operations and execution operations.

In summary, we have now arrived at three interlocked projections or partial models that seem to be useful to have, namely (1) a second-order data-structure model, which describes the system as a location structure; (2) a program-structure model which describes the system as data-flow between top-level modules, and a procedure-call structure that extends downward from there; and finally (3) an initialization model that describes how locations are filled with their contents. Additional projections or models have been proposed (for example the idea of a "purpose flow", due to Rich and Shrobe) but would extend the discussion too far for the present report.

Data base management

Before the paper ends, we should add two short remarks. One concerns the management of the integrated program/data structure. The proposals in the present section imply that the programming system should contain not only programs and data, but also descriptions of data, descriptions of programs, descriptions of their relationships, and so on. In order to use these proposals in practice, one must generalize conventional program management systems into data base management systems which treat procedures in the data base as a special case. (Sandewall, 1975) describes a first shot at such a data base management system.

The structure of the programming system

A LISP programming system itself takes some steps in the direction of conceptual programming. First, like any other interpreter it is data-driven with respect to names of elementary functions or procedures. But a modern LISP system also contains a large number of "handles" where the user can attach his own code or parameters to modify the system. It would be interesting to attempt to design a programming system using the methods proposed in this paper. Hopefully one would obtain a more consistently organized and more self-documenting system than is usually the case. Moreover, the models of the programming system might be useful for example when the user has to describe an initialization process where certain handles on the system are set in order to prepare ground for the continued initialization, for example when the status of certain characters (such as break characters) is changed.

PART IV

SUMMARY AND CONCLUSIONS

The present work has a strongly inductive character. Given a set of ideas about how one ought to think about a program and about programming, here characterized as "conceptual programming", I have studied one particular program in depth, partly in order to obtain suggestions for how the proposed "conceptual programming" method is to be realized. These are suggestions rather than empirically proven facts, since the study of one particular program would not provide sufficient evidence for the latter. They are also highly interdependent but I shall still venture to make a list of the conclusions.

Conceptual programming is an approach which contains two programming methods which are commonplace among users of LISP and similar languages already today, namely

- data-driven procedures (= dispatching), where procedures are called indirectly via data items;
- automatic advising, where the programmer first writes or generates a "skeleton" code, and then lets other procedures insert pieces of code into the skeleton.

It also contains a style of programming which is not in current use today, since it requires its own kind of programming system, but which seems sufficiently concrete that one could make some experiments with using it, namely

- insertive programming, where the program is represented by a hierarchical structure, plus a number of inserts of various kinds (additional pieces of code, information about data flow, etc.) which are located elsewhere but point into the structure to indicate where they belong at execution time.

In the very long range, conceptual programming is the ideal situation where the program is truly represented as a belief structure in the computer, and the programmer or tutor normally does not work with exhaustive listings of this structure, but only with specific extracts or projections which have been obtained for specific purposes.

The major conclusions which were obtained from the study of the simple program were:

- the location concept, where a location is a globally defined position which is described by a combination of identifiers that are understood by the user, and which may contain parameters, or a procedure, or certain dynamically changed data (for example catalogues).
- the importance of aggregates, i.e., structures which allow a dual description either as programs in very specialized languages, or as composite data structures that contain several locations.
- the usefulness of dynamic modification of aggregates, which may be thought of as self-modification in a program.
- the usefulness of second-order data-structure descriptions, where the information contained in conventional declarations, as well as other information about the same entities, is stored in the data base itself.
- a programming style, here called data-structure-based programming, where one specifies a tentative second-order data-structure model early in the design process, and then attempts to use combinations of entities in that model as names for locations as the program is developed. The purpose of this process is to obtain a system with a clear and self-documenting structure for both data and programs, and also to achieve economy of concepts and minimize the proliferation of "mnemonic" names. The support for this suggestion, besides its possible intuitive attraction, consists of a number of constructs in the sample program which would very likely have been designed in a more systematic way if the proposed programming style had been used.
- the second-order data-structure model is palatable only if it co-exists with a program-structure model, which on the top level (at least for

the present example) describes the data flow between a number of "data pools", and below that level, a calling-structure model.

- the data-flow model at the top of the program-structure model should not distinguish between "programs" and "data", since low-level program generation is common; since the distinction is irrelevant for the natural grouping of procedures and parameters into aggregates and into modules; and since files in this kind of programming system are most appropriately viewed as a kind of program.
- instead of the program/data distinction, there is a signficant distinction between "input" and "create" arrows. A procedure call may be considered as data input from the callee to the caller.
- besides the data-structure and the program-structure models, one also needs an initialization model. This model consists of two parts. The first part is a description of an interactive session, which is a kind of program which may call procedures and/or files (those two are entirely equivalent for this purpose). The session model is a skeleton in the sense of automatic advising. The second part of the initialization model is a set of inserts which for each location in the data-structure model specifies where in the initialization model there should be an operation that initializes this location.

ACKNOWLEDGEMENTS

It should be clear from the foregoing that many thanks are due to Dave McDonald, who let his program be subjected to this exercise, and also took a lot of time to discuss the program as well as the issues addressed in this paper. Several other researchers contributed with comments and ideas, particularly Anders Beckman (Uppsala), Carl Hewitt (MIT), Mats Nordström (Uppsala), Charles Rich, Howard Shrobe and Jerry Sussman (all MIT), and Terry Winograd (Stanford). The work reported here was mainly conducted at the MIT Artificial Intelligence Laboratory and the Stanford Artificial Intelligence Laboratory, both supported in part by the Advanced Research Projects Agency of the U.S. Department of Defense.

REFERENCES

Beckman, L., A. Haraldson, Ö. Oskarsson, and E. Sandewall (1975) A partial evaluator, and its uses as a programming tool. *Internal report DLU 74/34*, Department of Computer Science, Uppsala University, Sweden.

Dahl, O.-J. (1972) Hierarchical program structures, in O.-J. Dahl *et al., Structured Programming,* Academic Press.

Rich, Ch. and H. Shrobe (1974) Understanding LISP programs: Towards a programming apprentice. *Working paper 82,* MIT Artificial Intelligence Lab.

Sandewall, E. (1971) PCDB, a programming tool for management of a predicate-calculus-oriented data base. *Proc IJCAI2,* Computer Society, London.

Sandewall, E. (1973) Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs. *Proc IJCAI3*, Stanford Research Institute.

Sandewall, E. (1975) Ideas about management of LISP data bases. *Proc IJCAI4*, MIT.

Teitelman, W. (1974) *INTERLISP Reference Manual.* Xerox Palo Alto Research Center, Palo Alto, California.

Wegbreit, B. (1974) The treatment of data types in EL1. *CACM* 17.

Winograd, T. (1974) Five lectures on Artificial Intelligence. *Memo No. 246*, Stanford A.I. Laboratory.

Winograd, T. (1974) Breaking the complexity barrier, Stanford A.I. Laboratory.

Wirth, N. (1973) *Systematic Programming, an Introduction.* Prentice-Hall.